4.4BSD
オペレーティングシステムの設計と実装

目次

Chapter 2.1. 4.4BSD の設計の概要

2.1.1. 4.4BSD の機能とカーネル

4.4BSD カーネルは 4 つの基本機能を提供します。
それはプロセス、ファイルシステム、コミュニケーション、そしてシステムの起動です。 この節ではその 4 つの基本サービスのそれぞれについて この本で書かれていることを紹介します。

1. プロセスはアドレス空間上でのコントロールの流れを構成します。
   生成や終了やその他のプロセスをコントロールするための仕組みは 4
   章に述べます。システムは各プロセスの個別の仮想アドレス空間を
   多重化します。このメモリ管理については 5 章で議論します。
2. ファイルシステムとデバイスへのユーザインタフェースは似ているため、 6
   章ではそれらに共通する特徴について議論します。 7 章で説明するファイルシステムは、
   ディレクトリが木構造になった階層で組織された名前付きのファイルと、
   それらを扱うための操作からなります。
   ファイルはディスクのような物理的なメディア上に存在します。 4.4BSD
   はディスク上のデータ配置法をいくつかサポートしており、 それは 8 章の中で述べます。
   遠隔マシン上のファイルへのアクセスについては 9 章、
   システムにアクセスするために使われている端末とその動作については 10 章で扱います。
3. UNIX で古くから提供されている通信機構には、
   関連するプロセス間における単純で信頼性の高いバイトストリーム (11.1 節パイプを参照)および、
   例外イベントの通知 (4.7 節シグナルを参照)があります。 また、4.4BSD
   は汎用のプロセス間通信機構も備えています。 11 章に述べるこの通信機構は、
   ファイルシステムのものとは異なるアクセス機構を使用していますが、 一度接続が確立されれば、
   プロセスからはパイプと同じようにアクセスすることができます。 12
   章では汎用のネットワーク通信フレームワークについて扱っています。 これは通常、IPC
   機構の下位レイヤとして使われているものです。 13
   章では、ある特定のネットワークの実装について詳細に述べます。
4. いかなる実際のオペレーティングシステムにも、
   どのように起動するかというような運用上の話題があります。 14
   章では起動時や運用上の話題について述べます。

2.3 節から 2.14 節は 3 章から 14 章の内容を紹介するものです。
わたしたちは用語を定義し、基本的なシステムコールについて扱い、
開発の歴史について解説していきます。 そして最後に、中心となっている数多くの設計が、
どうやって選ばれたのかという理由を示します。

2.1.1.1. カーネル

カーネル はプロテクトモードで動作し、 すべてのユーザプログラムが基本的なハードウェア (たとえば CPU、ディスク、端末、ネットワーク接続機器) および ソフトウェアを構成するもの (たとえばファイルシステム、 ネットワークプロトコル) へのアクセスを解決します。
カーネルは基礎的なシステムの機能を提供します。
それはプロセスを生成して管理を行い、ファイルシステムへのアクセス機能や
コミュニケーション機能を提供します。 システムコール と呼ばれるこれらの機能は

ライブラリのサブルーチンとしてユーザプロセスに現れます。
これらのシステムコールはプロセスがこれらの機能に対して持っている 唯一のインタフェースです。
システムコール機構の詳細は、 システムコールを実行することで実現されているもの以外の、
いくつかのカーネル内機構を説明した 3 章で扱います。

従来のオペレーティングシステムの用語における カーネル とは、
オペレーティングシステムにサービスを追加する実装を行うために必要な
最小限の仕組みだけを提供する、 ソフトウェアの小さな核となる部分のことです。
同時代のオペレーティングシステムの研究、たとえば Chorus Rozier et al, 1988, Mach Accetta et al, 1986, Tunis Ewens et al, 1985, V Kernel Cheriton, 1988 では、
このカーネルという機能による区分が、さらに論理的に複数に分けられています。
ファイルシステムやネットワークプロトコルといったサービスは、
その核もしくはカーネルに対するクライアントアプリケーションプロセスとして 実装されています。

4.4BSD カーネルは複数のプロセスには分割されていません。 この基本設計の決定は UNIX の最初のバージョンで行われました。 Ken Thompson によって行われた初めの 2 つの実装では メモリマッピングがなく、
ユーザおよびカーネル空間はハードウェアによる分離が行なわれていませんでした Ritchie, 1988。
メッセージ伝達システムは、 実際に実装されたカーネルとユーザプロセスのモデルと
同じくらい容易に実装することが可能でした。
単純化と性能のためにモノリシックカーネルが選ばれました。
また、初期のカーネルは非常に小さいものでしたが、 ネットワークのような機能が追加されることで
次第に大きくなっていきました。 現在のオペレーティングシステムの研究の最先端では
そのようなサービスをユーザ空間に置くことで カーネルの大きさを減らそうとする傾向にあります。

ユーザは通常、 シェルと呼ばれる コマンド言語インタプリタや、
追加されたユーザアプリケーションプログラムを通してシステムと対話します。
そのようなプログラムやシェルは、プロセスを使って実装されています。
それらのプログラムの詳細についてはこの本の範囲を超えますので、
ここではカーネルについてのみ、考えることにします。

2.3 節、2.4 節では 4.4BSD カーネルによって提供されるサービスや最近の設計の概要について扱います。
そして後の章では、 4.4BSD に現れるそれらのサービスの設計と実装の詳細を説明します。

2.1.2. カーネルの構成

この節では、2 つの側面から 4.4BSD カーネルの構成を見ていきます。

1. ソフトウェアの静的側面、つまりカーネルを構築するためのモジュール
   によって提供された機能性による分類
2. その動的な機能、つまりユーザに提供されるサービスによる分類

カーネルの大部分は、システムコールを通してアプリケーションが
アクセスするためのシステムサービスを実装しています。 4.4BSD
では、このソフトウェアは次のような構成になっています。

- 基礎的なカーネル機能: タイマーおよびシステム時計の取り扱い、 記述子管理、そしてプロセス管理
- メモリ管理のサポート: ページングとスワッピング

- 汎用のシステムインタフェース: I/O、コントロール、 記述子によって実現される多重化操作
- ファイルシステム: ファイル、ディレクトリ、パス名の解釈、ファイルロック、 I/O バッファ管理
- 端末の取り扱いのサポート: 端末のインタフェースドライバと ラインディシプリン
- プロセス間通信機能: ソケット
- ネットワーク通信への対応: 経路制御等の通信プロトコル、一般的なネットワーク機能

表 *1. 4.4BSD* カーネルにおける機種非依存なソフトウェア

分類	コード行数	カーネル内での割合
ヘッダ	9,393	4.6
初期化部分	1,107	0.6
カーネルの機能	8,793	4.4
汎用のインタフェース	4,782	2.4
プロセス間通信	4,540	2.2
端末の取り扱い	3,911	1.9
仮想メモリ	11,813	5.8
vnode 管理	7,954	3.9
ファイルシステムネーミング	6,550	3.2
FFS	4,365	2.2
ログ構造化ファイルシステム	4,337	2.1
メモリベースのファイルシステム	645	0.3
cd9660 ファイルシステム	4,177	2.1
その他のファイルシステム (10)	12,695	6.3
ネットワークファイルシステム	17,199	8.5
ネットワーク通信	8,630	4.3
インターネットプロトコル	11,984	5.9
ISO プロトコル	23,924	11.8
X.25 プロトコル	10,626	5.3
XNS プロトコル	5,192	2.6
機種非依存部分の総計	162,617	80.4

これらのカテゴリのソフトウェアのほとんどは機種非依存であり、しかも
異なるハードウェアアーキテクチャに移植できるものです。

カーネルの機種依存部分は本流のコードから分離されています。
特に、機種依存しているコードのどれを取っても、
特定のアーキテクチャのためのコードを含んでいません。
機種に依存する機能が必要なときには、機種に依存しないコードは
機種依存のコード内にあるアーキテクチャ依存の関数を呼び出します。
機種依存であるソフトウェアは次のものを含んでいます。

- 低レベルなシステム起動のための動作
- トラップおよびフォールトの扱い
- プロセスの動作状態に関する低レベルな操作
- ハードウェアデバイスの設定と初期化
- I/O デバイスのランタイムサポート

表 *2. 4.4BSD* カーネル内にある *HP300* 用の機種依存部分

分類	コード行数	カーネル内での割合
機種依存部分のヘッダ	1,562	0.8
デバイスドライバのヘッダ	3,495	1.7
デバイスドライバのソースコード	17,506	8.7
仮想メモリ	3,087	1.5
他の機種依存部分	6,287	3.1
アセンブリ言語で書かれたルーチン	3,014	1.5
HP/UX 互換機能	4,683	2.3
機種依存部分の総計	39,634	19.6

4.4BSD カーネルにおける機種非依存なソフトウェア は HP300 用 4.4BSD カーネルを構成するソフトウェアのうち、 機種非依存の部分をまとめたものです。 2 列目の数値は C 言語のソースコード、ヘッダファイル、そして アセンブリ言語のものを表しており、アセンブリ言語のものは 2% 以下しかありません。 また、4.4BSD カーネル内にある HP300 用の機種依存部分 の統計が示しているように、 機種依存のソフトウェアは、HP/UX やデバイスサポートを除いて、 カーネルのうちわずかに 6.9% しかありません。

カーネルのごく小さな部分だけがシステムの初期化に専念します。 このコードはシステムが 起動するときに用いられ、 カーネルがハードウェアやソフトウェアの環境構築をするための基本部分と なります (14 章参照)。 (制限された物理メモリを持つものには特に) オペレーティングシステムの中にはこれらの機能が実行された後にそのソフトウェアを廃棄してしまうか、 上から覆ってしまうものもあります。 4.4BSD カーネルは、起動するために要したコードのためのメモリを再利用しません。それは通常の機種ではカーネルのリソースのうち 0.5% に過ぎないからです。そして、起動のためのコードはカーネルの中の一部分に固まってはいません。
それは全体にわたって散在しており、初期化されたときに論理的に関連していた 場所に存在します。

2.1.3. カーネル サービス

カーネルレベルコードとユーザレベルコードの間の境界は、
基盤となるハードウェアで提供されるハードウェアレベルの保護機能によって 分離されています。カーネルは、ユーザプロセスにとってアクセスしにくい切り離された
アドレス空間で作動します。特権のある操作 — たとえば I/O の開始や中央処理装置 (CPU) の停止 — は、カーネルだけが利用可能です。 アプリケーションはシステムコールを用いてカーネルにサービスを要求します。システムコールは二次記憶装置にデータを
書き込むような複雑な作業や、現在の日時を返すような単純な作業を

カーネルに実行させるために使用されます。 アプリケーションからは、
すべてのシステムコールが同期的に実行するように見えます。
つまりアプリケーションは、カーネルがシステムコールに関連した
動作をしているときには停止しています。 カーネルはシステムコール操作の一部を、
システムコールが戻った後に完了することもあります。 たとえば *write* システムコールは、
プロセスが待っている間に書き込むデータをユーザプロセスからカーネルバッファにコピーしますが、
通常、そのカーネルバッファがディスクに書き込まれる前にシステムコールから戻ります。

通常、システムコールは CPU の実行モードおよび現在のアドレス空間マッピングを変更する
ハードウェアトラップとして実装されています。
ユーザによって与えられたシステムコール中のパラメータは、
使用される前にカーネルによって検証されます。 そのようなチェックはシステムの完全性を保証します。
カーネルへ渡されたパラメータはすべてカーネルのアドレス空間にコピーされます。
これは、システムコールの副作用により検証されたパラメータが 変更されないことを保証するためです。
システムコールの戻り値は、カーネルによってハードウェアレジスタ中に返されるか、
あるいはユーザが指定したメモリアドレスにコピーされる値で返されます。
カーネルへ渡されたパラメータのように
結果を戻すためにアプリケーションによって指定されたアドレスは、
それらが確実にアプリケーションのアドレス空間の一部である、
ということが検証されなければなりません。
システムコールを処理する間にカーネルがエラーに遭遇した場合、カーネルは
ユーザにエラーコードを返します。 C プログラミング言語においては、このエラーコードが大域変数
*errno*に格納されます。また、システムコールを 実行した関数は -1 の値を返します。

ユーザアプリケーションとカーネルは、互いに独立して動作します。 4.4BSD は I/O
コントロールブロックや、 オペレーティングシステムに関連するその他のデータ構造体を
アプリケーションのアドレス空間に格納しません。 ユーザレベルのアプリケーションはそれぞれ、
実行のための独立したアドレス空間を提供されます。
カーネルは、たとえば別のプロセスが走っている間そのプロセスを停止させ、
関係するプロセスに見えないようにするというような、 状態変更のほとんどを実現しています。

2.1.4. プロセス管理

4.4BSD はマルチタスク環境をサポートしています。 実行されたそれぞれのタスクまたはスレッドは
プロセスと呼ばれています。 4.4BSD のプロセスのコンテキストは、
アドレス空間の内容とランタイム環境を含むユーザレベルの状態と、
スケジューリングのパラメータやリソース制御、識別情報を含む
カーネルレベルの状態から構成されています。 コンテキストにはカーネルがプロセスにサービスを
提供する際に使用するすべてが含まれています。 ユーザはプロセスを生成し、その実行を制御し、
プロセスの実行状態が変化したときに通知を受け取ることができます。 すべてのプロセスにはプロセス
ID (PID) と呼ばれる一意の値が割り当てられます。
この値はカーネルがユーザに実行状態の変化を報告するときにプロセスの身元を確認したり、
ユーザがシステムコールを実行するために参照する際に使用されます。

カーネルは他のプロセスのコンテキストを複製してプロセスを生成します。 新しく生成されたプロセスを
元の親プロセスの 子プロセスと呼びます。
プロセス生成時に複製されたコンテキストは、ユーザレベルのプロセスの実行状態と
カーネルが管理しているプロセスのシステム状態の両方を含んでいます。
カーネルの状態に関する重要な構成要素については、4 章で解説しています。

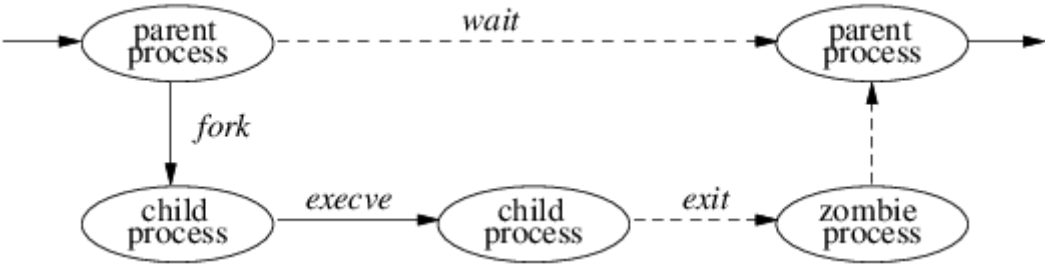

図 1. プロセスのライフサイクル

プロセスのライフサイクルではプロセスのライフサイクルを示しています。 プロセスは *fork* システムコールを用いて、 元のプロセスのコピーとして新しいプロセスを生成することができます。 *fork* は呼び出されると 二度戻ります。一方は親プロセスに子プロセスのプロセス ID を返し、もう一方は子プロセスに 0 を返します。
プロセスの親子関係はシステム上のプロセスの組に階層構造をもたらします。
新しく生成されたプロセスはファイル記述子やシグナルハンドラの状態、
メモリレイアウトのような親が持っているリソースすべてを共有します。

親のコピーとして生成された新しいプロセスであっても、
別のプログラムをロードし実行することでより便利で特有の動作をすることもできます。 プロセスは *execve* システムコールを用いることで、 別のプログラムのメモリイメージで自分自身を上書きして、新しい引数の組をその新しく作成したイメージに引き渡すことができます。
引数の一つは、システムで認識されるフォーマット
(バイナリ実行ファイルや指定されたインタプリタプログラムの起動を促すファイル)
をしたファイルの名前です。

プロセスは *exit* システムコールを実行することで、 親プロセスに 8 ビットの exit ステータスを送信して終了することができます。 もしプロセスが 1 バイト以上の情報を親プロセスに伝えたい場合には、 パイプやソケット、または仲介ファイルを用いてプロセス間通信チャネルをセットアップする必要があります。 プロセス間通信については 11 章で大きく取り上げています。

プロセスは、*wait* システムコールを用いて子プロセスのいずれかが終了するまで実行を中断することができ、 wait システムコールは終了した子プロセスの PID と終了ステータスを返します。
親プロセスは、子プロセスが終了または異常終了したときのシグナルによる通知のされ方を調整できます。 *wait4* システムコールを使用することで、
親プロセスは子プロセスの終了を引き起こしたイベントについての情報と、
子プロセスが生存期間の間に消費したリソースについての情報を取得することができます。
もし親プロセスが先に終了したためにリソースがオーファンド (親のない状態)になってしまった場合、カーネルは *init* という特別なプロセスにその子プロセスの終了ステータスが渡されるよう調整します。これについては 3.1 節および 14.6 節を参照してください。

5 章では、カーネルがどのようにしてプロセスを生成し 消滅させるかについての詳細を述べています。

プロセスはプロセス優先度というパラメータに従って 実行をスケジュールされます。
この優先度はカーネルベースのスケジューリングアルゴリズムによって管理されています。
スケジューリングの優先度全体に重みづけする特別なパラメータ (*nice*)によって、ユーザはプロセスの実行優先度に影響を与えることができますが、
カーネルのスケジューリングポリシに従って、基本となる CPU リソースを共有する必要があります。

2.1.4.1. シグナル

システムはプロセスに送ることができる　　シグナルのセットを定義しています。　　4.4BSD
におけるシグナルはハードウェア割り込みをモデルとしています。
プロセスはユーザレベルのサブルーチンをシグナルが送られるべき　　ハンドラとして指定できます。
シグナルが発生して、それがハンドラによって捕捉されている間は
さらなるシグナルの発生はブロックされます。
シグナルを捕捉することで、現在のプロセスのコンテキストを保存し、
ハンドラを実行するための新たなコンテキストを構築することになります。
シグナルがハンドラに伝わると、そのハンドラはプロセスをアボートさせたり、
(おそらく大域変数に値を設定した後で)　　実行中のプロセスに戻ることもできます。
ハンドラから戻ると、そのシグナルはブロックされなくなり、　　発生する　　(そして捕捉される)
ことが再び可能になります。

また、プロセスはシグナルを無視することや、
カーネルで定義されているデフォルトの動作を行なうように指定することができます。
ある種のシグナルのデフォルトでの動作はプロセスを終了させることです。
このような場合の終了は、事後のデバッグに使用できるようにその時のプロセスのメモリイメージを含んだ コアファイルの生成を伴います。

いくつかのシグナルは捕捉することも無視することもできません。
そのシグナルは、暴走したプロセスを停止させる　*SIGKILL*　や、　ジョブコントロールシグナルである *SIGSTOP* です。

プロセスはシグナルを特別なスタックに伝達させることも選択できます。
これにより、洗練されたソフトウェアスタック操作が可能です。
たとえば、コルーチンをサポートしている言語では
それぞれのコルーチンにスタックを提供する必要があります。　　その言語の実行システムは、4.4BSD
で提供される単一のスタックを分割することで、　　これらのスタックを割り当てることができます。
もしカーネルが独立したシグナルスタックをサポートしていない場合、
それぞれのコルーチンに割り当てられた領域を
シグナルの捕捉に必要な分だけ拡張しなければなりません。

すべてのシグナルは、同じ優先度を持っています。
もし複数のシグナルが同時に未処理となっている場合は、　　シグナルの届く順序は実装に依存します。
シグナルハンドラは、そのシグナルがブロックされるようにして実行しますが、
他のシグナルは依然発生可能です。　　このメカニズムにより、プロセスはコードのクリティカルな部分を特定のシグナルの発生に対して保護することができるのです。

シグナルの設計と実装の詳細は、4.7 節で解説しています。

2.1.4.2. プロセスグループとセッション

複数のプロセスを組織してプロセスグループが作られます。
プロセスグループは端末へのアクセスの制御や
関係プロセスの集合にシグナルを送る手段を提供するのに使用されます。
プロセスは親プロセスからプロセスグループを引き継ぎます。
プロセスが自分自身または自分の子孫のプロセスグループを変更できるようにする
メカニズムをカーネルは提供しています。　　新しいプロセスグループを作成することは簡単です。
新しいプロセスグループの値はたいてい 作成したプロセスのプロセス ID となります。

プロセスグループにおけるプロセスの集合は、ジョブと呼ばれることがあり、
シェルのような高レベルのシステムソフトウェアで操作されます。
シェルによって生成されるよくある類のジョブは、いくつかのプロセスをパイプでつないだ
パイプラインで、最初のプロセスの出力が 2 番目の入力となり、 2 番目の出力が 3 番目の入力となり、4 番目も同様に... というものです。 シェルはパイプラインの各段階においてプロセスを fork して、これらすべてのプロセスを別個のプロセスグループにおくことで このようなジョブを生成します。

ユーザプロセスは、単独のプロセスに送る場合と同様に
プロセスグループのそれぞれのプロセスにまとめてシグナルを送ることができます。
指定されたプロセスグループに属するプロセスが
そのプロセスグループに影響するソフトウェア割り込みを受け取ると、
それによってプロセスグループは実行を中断や再開をしたり、
割り込みを受けたり、終了させられたりします。

端末にはプロセスグループ ID が割り当てられています。 この ID は、端末に関連づけられたプロセスグループの ID が通常セットされます。
ジョブコントロール機能を持つシェルは、同じ端末に関連づけされた
プロセスグループを多数作成することができます。
その端末は、これらのプロセスグループに属するプロセスの制御端末となります。
プロセスは、端末のプロセスグループ ID とそのプロセスのプロセスグループ ID が一致したときのみ、制御端末を記述子から読むことができます。 もしプロセスグループ ID が一致していなければ、プロセスがその端末から読み込もうとする際にブロックされます。 端末のプロセスグループ ID を変更することで、 シェルはいくつかの異なるジョブの間で端末を調停することができます。
この調停はジョブコントロールと呼ばれ、 プロセスグループとともに 4.8 節で解説しています。

関連するプロセスの集合をプロセスグループとしてまとめることができるのと同じように、
プロセスグループの集合をセッションとしてまとめることができます。
セッションのおもな用途は、デーモンプロセスとその子プロセスに対して隔離した環境を作り出したり、
ユーザのログインシェルとそのシェルが作り出すジョブをひとまとめにすることです。

2.1.5. メモリ管理

それぞれのプロセスはプロセスごとのプライベートアドレス空間を持っています。
アドレス空間は、最初に論理的な3つのセグメントに分割されます: テキスト、 データ、およびスタックです。 テキストセグメントは読み出し専用で、プログラムの命令を含んでいます。
データ及びスタックセグメントは読み取り書き込みともに可能です。 データセグメントには初期化されているデータと初期化されていないデータがあるのに対し、
スタックセグメントはランタイムスタックを保持します。
ほとんどのマシンでは、プロセスが実行するとともに、
カーネルによってスタックセグメントは自動的に拡張されます。
プロセスはシステムコールによりデータセグメントを拡張する事が可能ですが、
セグメントの内容がファイルシステムからのデータである場合、あるいは
デバッグ時に限りプロセスはそのテキストセグメントのサイズを変更することができます。
子プロセスのセグメントの初期の内容は親プロセスのセグメントのコピーです。

プロセスアドレス空間の全内容はプロセスが実行するのには必要がありません。
プロセスがメインメモリにおいて保持されていないアドレス空間の一部を参照する場合、
システムはメインメモリーからメモリの中の必要な情報を ページ につけます。
システムリソースが不足する場合、システムは利用可能な資源を維持するために2レベルのアプローチを

します。 適度の量のメモリが利用可能な場合でこれらの資源が最近使用されていない場合、システムはプロセスからメモリリソースを解放します。
メモリー不足が深刻だった場合、システムはプロセスの全情況を２次キャッシュの スワップに頼ります。 ページング と スワップ の交換はシステムによって行われた、プロセスに有効です。プロセスは実行援助として予期された将来のメモリ利用についてシステムに助言するかもしれません。

2.1.5.1. BSDメモリ管理設計の決定

疎の広いアドレス空間のサポート、 メモリマップファイル、共有メモリは、 4.2BSDに要求されたものの一つでした。
独立したプロセス群がプロセスのアドレス空間をファイルにマッピングし、 それの共有を可能にする *mmap* と呼ばれるインタフェースが規定されました。
複数のプロセスが同じファイルにプロセスのアドレス空間をマッピングした場合、
一つのプロセスがファイルにマッピングされたアドレス空間の一部分に対して
加えた変更は、通常のファイルがそうであるのと同様、
同じ部分をマッピングしている他のプロセスにも反映されます。 しかし結局、4.2BSDは *mmap* インタフェースを含まない形でリリースされました。
これはネットワークのような他の機能を実現する方が重要で、 時間的な余裕がなかったからです。

mmap インタフェースの開発は、4.3BSD の作業の間も続けられました。 40社を超える会社と研究グループが、 Berkeley Software Architecture Manual McKusick et al, 1994 に記載されたアーキテクチャの改訂版を策定する議論に参加し、
いくつかの企業はその改訂版のインタフェースを実装しました Gingell et al, 1987。

しかし、またもや時間的な問題により 4.3BSD への mmap インタフェースの実装は見送られました。もちろん既存の 4.3BSD 仮想記憶システムにそのインタフェースを組み込むことは可能だったのですが、4.3BSD の仮想記憶システムの実装は 10年近く前のものであったため、開発者たちはそれを組み込まないことに決定したのです。 4.3BSDの仮想記憶システムはローカルに接続されたディスク装置は高速・大容量・安価で、
コンピュータのメモリは小容量・高価であるという仮定に基づいて設計されており、
そのため、その設計はメモリ利用量を節約できる代わりに
余分なディスクアクセスを生成してしまうものでした。 また、この実装は VAXのメモリ管理ハードウェアに強く依存するもので、
他のコンピュータアーキテクチャへの移植が困難でした。
最後にもう一つ付け加えるなら、この仮想記憶システムは
現在普及がすすみ重要になってきている密結合マルチプロセッサに
対応するように設計されていなかったのです。

古い仮想記憶システムの実装を改良しようという試みは、
ますます失敗が運命づけられたように思われました。
その一方で、完全に新しい設計は大容量メモリを利用し、 ディスクへのデータ転送を低減し、マルチプロセッサで動作することができる能力を持っていました。 その結果、仮想記憶システムは 4.4BSD で完全に置き換えられることになったのです。 4.4BSD 仮想記憶システムは Mach 2.0 VM システム Tevanian, 1987 をベースに、Mach 2.5 と Mach 2.0 の改良を採り入れたものです。この実装は、 メモリ共有の効率が良く機種依存部分と機種非依存部分がきれいに分離されていて、(現在は使われていませんが) マルチプロセッサに対応しているという特徴を持っています。
各プロセスは自分のアドレス空間のあらゆる部分をファイルにマッピングすることができ、
互いに同一のファイルにアドレス空間をマッピングすることで、
プロセス間でアドレス空間の一部を共有することが可能になりました。

一つのプロセスが加えた変更は他のプロセスのアドレス空間にも反映され、
マッピングされたファイル自身にも書き込まれます。
また、プロセスはファイルをプライベートマッピングすることも可能です。
プライベートマッピングとは、プロセスが加えた変更が、
そのファイルをマッピングしている他のプロセスから見えないようにしたり、
ファイル自身に書き戻されないようにするものです。

仮想記憶システムの抱えるもう一つの問題は、
システムコールが発行された時にカーネルに情報を渡す方法です。 4.4BSD
では、常にプロセスのアドレス空間からカーネル内のバッファに データをコピーしていました。
大容量のデータを転送する読み書き操作が発生することを考えると、
このコピーの実行には時間がかかる可能性があります。 コピーを実現するもう一つの方法として、
プロセスのメモリをカーネル内に再マッピングする方法があります。 しかし 4.4BSD カーネルは、
以下の理由から常にデータをコピーします。

- ほとんどの場合ユーザデータはページ境界にアラインされていませんし、
  ハードウェアページ長の倍数でもありません。
- そのページをプロセスが破棄してしまうと、 カーネルがページを参照できなくなってしまいます。
  プログラムの中には、カーネル内のバッファに
  書かれたデータが残っていることを想定しているものがあります。
- (現在の 4.4BSD セマンティクスで可能なように) プロセスがページのコピーを持てる場合、
  そのページは必ずコピーオンライト*(copy-on-write)* になっています。
  コピーオンライトのページとは、
  読み込み専用に設定することで書き込みに対する保護機能を有効化したページのことです。
  プロセスがそのページを変更しようとするとカーネルは書き込み例外を検出します。
  その際カーネルは、プロセスが変更できるようにそのページのコピーを作成します。
  残念ながら、プロセスは通常すぐに出力バッファに新しいデータを書こうとするため、
  結局データのコピーが発生してしまいます。
- ページが新しい仮想メモリアドレスに再マッピングされる際、
  ほとんどのメモリ管理ハードウェアでは、
  ハードウェアアドレス変換キャッシュの一部を破棄する必要があります。
  多くの場合、このキャッシュの破棄は時間がかかるため、 4 から 8
  キロバイトより小さいデータブロックに対しては、
  コピーするよりも再マッピングする方が実質的に遅い、 という結果となります。

メモリマッピングの最も大きな目的は、 巨大なファイルへのアクセスと、
プロセス間の大容量のデータ転送という要求に応えることです。 *mmap* インタフェースは、
両方の要求をコピーを行なうことなく実現する一つの方法を提供します。

2.1.5.2. カーネル内部のメモリ管理

カーネルは一つのシステムコールの間だけ必要とされるメモリの割り当てを頻繁に行ないます。
ユーザプロセスではおそらく、
そのような短期間使われるメモリはランタイムスタックに割り当てられるでしょう。
カーネルのランタイムスタックには上限があるため、
小さめのメモリブロックだとしてもスタックにメモリを割り当てることはできません。 そのため、
そのようなメモリはもっと動的な機能を用いて割り当てる必要があります。
たとえば、システムがパス名の解釈を行なう場合、 パス名を保持するために 1

キロバイトのバッファを割り当てる必要があります。
しかしメモリブロックは一つのシステムコールよりも 長く持続していなければならないため、
スタックに空きがあったとしても、そこに割り当てることはできないでしょう。
こういう例の一つに、ネットワークが接続されている間維持している必要がある
プロトコル制御ブロックがあります。

カーネル内の動的なメモリ割り当てに対する需要は、 サービスが追加されるにつれて増加しています。
汎用のメモリアロケータがあれば、 カーネル内部のコードを書く際の複雑さを低減することができます。
そのため 4.4BSD カーネルでは、システムのあらゆる場面で利用可能な
単一のメモリアロケータを備えています。 これは、
アプリケーションプログラム用のメモリ割り付けを実現するために C ライブラリルーチンに含まれている
malloc と *free* と類似したインタフェースを持っています McKusick & Karels, 1988。
この割り付けルーチンは C ライブラリインタフェースと同様、
引数として必要なメモリサイズを指定します。 割り当てるメモリサイズの上限はありませんが、
割り当てられるのは物理メモリであり、ページではありません。
メモリ解放ルーチンは解放するメモリへのポインタを引数にとります。
その際、解放するメモリサイズを指定する必要はありません。

2.1.6. I/O システム

基本的な UNIX の I/O システムモデルは、ランダムアクセスおよび
シーケンシャルアクセスの可能なバイト列です。 通常の UNIX ユーザープロセスには、
アクセスメソッド や コントロールブロック は存在しません。

I/O にさまざまなレベルの構造を期待するプログラムは各種ありますが、 カーネルは I/O
に構造を課しません。 たとえば、テキストファイルは改行文字 (ASCII LF 文字) で区切られた ASCII
文字の行の集まりですが、 カーネルはそのような構造を関知しません。
ほとんどのプログラムにとって、 このモデルはデータバイトのストリームもしくは *I/O* ストリーム
にすぎません。 このような単一のデータ構造が、UNIX のツールベースのアプローチ (tool-based
approach) を可能にしているのですKernighan & Pike, 1984。
あるプログラムの出力ストリームは、他のほとんどのプログラムの入力
ストリームとしてそのまま与える事ができます (このような伝統的な UNIX の I/O ストリームを、Eighth
Edition のストリーム I/O システムや、 System V Release 3 の STREAMS
と混同すべきではありませんが、 どちらのストリームも伝統的な I/O
ストリームと同じようにアクセスすることが可能です)。

2.1.6.1. 記述子と I/O

UNIX のプロセスは、I/O ストリームを参照するのに 記述子*(descriptor)*を使用します。 記述子は *open*
または *socket* システムコールにより取得される符号無しの小さな整数です。 *open*システムコールは、
引数にファイル名および許可モードをとり、 それぞれ開くファイルおよび、モード
(読み込み、書き込みまたは読み書き) を指定します。 *open*
システムコールは、新しい空のファイルの作成にも使用できます。 *read*および
*write*システムコールを記述子に対して使用し、 データの転送を行います。
*close*システムコールは、任意の記述子を開放します。

記述子は、カーネルでサポートされるオブジェクトを表します。 4.4BSD
では、ファイル、パイプ、ソケットの 3 つのオブジェクトを 表すことができます。

- ファイルは、少なくとも 1 個の名前を持つバイト列です。
  ファイルは、すべての名前を明示的に削除し、
  その記述子を持つすべてのプロセスが消滅するまで存在します。 プロセスは、*open* システムコールにより、 指定されたファイル名を持つファイルのファイル記述子を取得します。 I/O デバイスはファイルとしてアクセスされます。
- パイプとは、 ファイルと同じくバイト列ですが I/O ストリームとしてのみ使われ、単一方向にのみ使われます。 パイプには名前がないので、*open* システムコールでは開くことができません。 パイプを開くには、*pipe* システムコールを使用します。 *pipe*システムコールは 2 つの記述子を返します。 ひとつの記述子に入力されたデータは、もう一方の記述子にそのまま順序を変えずに出力されます。 名前付きパイプ (FIFO) も使用できます。 名前があるのでファイルシステム上に配置され、 *open* システムコールでアクセスできる以外は、パイプと同一の機能を持ちます。 FIFO を使用してプロセス間通信を行いたい場合は、片方のプロセスが FIFO を書き込み用に開き、 もう片方では読み込み用に開きます。
- ソケットは、 プロセス間通信のために使用されるオブジェクトで、ソケットを参照する記述子を持つプロセスが存在する間のみ存在します。 ソケットは *socket* システムコールで作成します。 *socket* システムコールは、 作成したソケットの記述子を返します。 さまざまな通信方法を実現するために、 各種のソケットがあります。 たとえば、信頼性の高いデータ転送を目的としたソケット、 メッセージの順番を保持するソケット、メッセージの境界を保護するソケットなどがあります。

4.2BSD でソケットが導入されるまで、 パイプはファイルシステムを用いて実装されていました。 4.2BSD 以降では、ソケットを使用して実装されています。

カーネルはそれぞれのプロセスの記述子テーブルを保持しており、
記述子の外部表現を内部表現に変換するために用いられます
(記述子そのものはこのテーブルへのインデックス値にすぎません)。
記述子テーブルは、親プロセスから子プロセスに継承されます。
そのため、記述子の参照先も同じく継承されます。 記述子を得るためには、オブジェクトを開いたり、作成したりする以外に、 このような親プロセスからの継承による方法があります。 また IPC ソケットを使用すれば、 同一マシン上で動作している無関係なプロセス間で、記述子のやりとりが可能です。

すべての有効な記述子は、 オブジェクトの先頭からの位置を ファイルオフセットとしてバイト単位で保持しています。 読み込みおよび書き込み動作は、このオフセット位置から行われ、 データが転送される毎にオフセットの位置は更新されます。 ランダムアクセスを許可しているオブジェクトの場合、 ファイルオフセットは、*lseek* システムコールを利用して移動することもできます。
通常のファイルやある種のデバイスはランダムアクセス可能です。
パイプ、ソケットはランダムアクセスできません。

プロセスが終了すると、 カーネルはそのプロセスに使用されていたすべての識別子を回収します。
プロセスがオブジェクトへの参照を保持したまま終了した場合は、
オブジェクトマネージャに通知し、ファイルの削除、
ソケットの開放などの必要なクリーンアップを行わせます。

2.1.6.2. 記述子の管理

ほとんどの場合、プロセスが起動されると、 3 つの記述子がすでに開かれています。 それらの記述子は、0、1、2 で、それぞれ一般的には、 標準入力、 標準出力、 標準エラー出力

として知られています。 通常これらの識別子は、
ログインプロセスによりユーザの端末に割り当てられています (14.6 節参照
)。すなわち、キーボードからの入力を標準入力として受け取り、
標準出力への出力は端末の画面に表示されます。
標準エラー出力もエラー出力用に書き込み用に開かれていますが、
通常の出力には標準出力が利用されます。

これらの記述子を (他の記述子も) 端末以外のオブジェクトに割り当てることも可能です。
このような割り当てを、 *I/O* リダイレクトと呼びます。 すべての標準シェルでは、ユーザが I/O
リダイレクトを行うことができます。 記述子 1 (標準出力) を閉じ、 指定したファイルを記述子 1
として開くことで、 シェルは出力をファイルに送ることができます。 同様に、記述子 0 を閉じ、
指定したファイルを開くことで、 ファイルから標準入力を受け取るようにできます。

パイプは、プログラムの変更をまったく行わず (再リンクも必要ありません)、あるプログラムの出力を
他のプログラムに入力することを可能にします。 出力側のプログラムの記述子 1 (標準出力)
は、端末出力の代わりにパイプの入力記述子に割り当てられます。 同様に入力側のプログラムの記述子 0
(標準入力) は、 端末からのキーボード入力ではなくパイプの出力記述子に割り当てられます。

open、 *pipe*、 *socket* システムコールは、新しい記述子を生成し、
使用できる最も小さい番号を割り当てます。
パイプを動作させるためには、そのように生成された記述子を 0 や 1
にマップする仕組みが必要になります。 *dup* システムコールは、
同一のファイルテーブルエントリを指す記述子のコピーを作成します。
新しい記述子も同じく使用可能な最小の番号が使われるため、 *dup*システムコールを使用して、
必要なマップを行えます。 ただ、記述子 1 が必要な場合でも、記述子 0 が既に閉じられていると、
記述子 0 が割り当てられてしまいますので注意が必要です。 この問題を避けるため、 *dup2*
システムコールがあります。 *dup* に引数が 1 つ追加され、
割り当てたい記述子の番号を指定することができます (もし、指定された番号の記述子が使用中の場合、
dup2 は、まずその記述子を閉じたのち、 再割り当てします)。

2.1.6.3. デバイス

ハードウェアデバイスはファイル名を持ち、通常のファイルと
同一のシステムコールでアクセスできます。カーネルは、 デバイス特殊ファイル や
特殊ファイルを区別し、 参照しているデバイスを特定できますが、
ほとんどのプロセスにとって、このような区別は必要ありません。
端末、プリンタ、テープデバイスは、4.4BSD のディスクファイルと同様、
バイト列としてアクセスされます。そのため、デバイス依存部分や特殊部分は、
可能な限りカーネルに隠蔽され、さらにカーネル内でも、
それらの大部分がデバイスドライバ内に分離されています。

ハードウェアデバイスは、 構造を持つデバイスと 構造を持たないデバイスに分けられます。 それぞれ、
ブロックデバイス、 キャラクタデバイスと呼ばれます。
それらのデバイスファイルへのアクセスは、カーネル内の デバイスドライバ
と呼ばれるソフトウェアモジュールによって処理されます。
ほとんどのネットワーク通信ハードウェアデバイスは、 ファイルシステム上に特殊ファイルを持たず、
プロセス間通信機能によってのみアクセスできます。 それは、*raw-socket*の方が特殊ファイルより、
より自然なインタフェースを提供できるためです。

典型的なブロックデバイス (構造を持つデバイス) としては、 ディスク、磁気テープがあげられますが、

ほとんどのランダムアクセスデバイスがそれに該当します。 カーネルは、読み込み-変更-書き込みに対してバッファリングを提供し、 通常ファイルと同様の、完全なバイトアドレス指定のランダムアクセスを提供します。
ファイルシステムは、ブロックデバイス上に構築されます。

構造を持たないデバイスは、 ブロック構造をサポートしないデバイスで、通信線、ラスタプロッタ、バッファのない磁気ディスクやテープなどです。 構造を持たないデバイスは通常、 大容量のブロック I/O 転送をサポートします。

構造を持たないファイルはキャラクタデバイスと呼ばれます。
これは最初に実装されたこの種類のデバイスが、 端末デバイスドライバだったからです。
このようなデバイスに対するカーネルのインタフェースは、
他のブロック構造を持たないデバイスに対しても有用であることが証明されました。

デバイス特殊ファイルは、 *mknod*システムコールにより作成されます。 *ioctl*システムコールは、特殊ファイルに対応するデバイスのパラメータを操作するのに使われます。
このシステムコールは、他のシステムコールに新たな機能を追加せずに、
デバイスの特殊な機能を操作することを可能にします。 たとえば、*ioctl*を使用して、終了マークをテープデバイスに書き込むことができます。 *write* に変更を加えたり、特殊なバージョンを用意する必要はありません。

2.1.6.4. ソケット IPC

4.2BSD カーネルはソケットを利用して、パイプより柔軟な IPC 機能を導入しました。ソケットは、ファイルやパイプと同様、
記述子により参照される、通信の末端点です。 2 つのプロセスがそれぞれ、ソケットを作成して接続することにより、
信頼性の高いバイトストリームを作成できます。
接続されれば、それぞれのプロセスは、パイプと同じように、
読み込み書き込みをソケットに対して行えます。
ソケットの透明性により、カーネルはプロセスの出力を、
別のマシン上のプロセスの入力に送ることも可能です。 パイプとソケットの大きな違いは、パイプは共通の親プロセスが設定する必要があるのに対して、 ソケットはまったく無関係の (異なるマシン上で動作する) プロセス間でも使用できる点です。

System V は、FIFO もしくは名前付きパイプと呼ばれるローカルプロセス間通信の仕組みを備えています。 FIFO はファイルシステム上のオブジェクトとして現われ、
パイプと同様な方法でオープンし、データを送ることができます。 そのため、FIFO は共通の親プロセスによって設定される必要はなく、
プロセス同士が起動し動作開始してから接続することが可能です。 しかしソケットとは異なり、異なるマシン上で動作するプロセスに対しては使用できません。 4.4BSD で、FIFO が実装されているのは、 POSIX.1 標準に準拠するためのみです。 FIFO の機能は、ソケットの機能の一部になっています。

ソケット機構を実現するには、 伝統的な UNIX の I/O システムコールに名前付けや接続機能を追加する必要がありました。
開発者は、既存のインタフェースへの拡張は既存のシステムコールが変更なしに使用できる範囲にとどめ、 追加機能を扱う新しいインタフェースを設計しました。
バイトストリーム型の接続の読み込み書き込みを行う *read*と *write* システムコールに加え、

ネットワークダイアグラムのような宛名付きメッセージを読み込むため、新たに 6 つのシステムコールが追加されました。メッセージ書き込み用の *send*、*sendto*、*sendmsg* システムコールと、メッセージの読み込み用の *recv*、*recvfrom*、*recvmsg* システムコールです。考え直して見ると、それぞれの読み書き用のシステムコールのうち最初の 2 つは次のシステムコールの特殊な場合であるので、*recvfrom* と *sendto* システムコールは、それぞれ *recvmsg* と *sendmsg* のライブラリインタフェースとし て追加すべきだったかも知れません。

2.1.6.5. Scatter/Gather I/O

既存の *read* および *write* システムコールに加え、4.2BSD で scatter/gather I/O 機能が導入されました。scatter 入力は *readv* システムコールによって行われ、複数の異なるバッファに対して単一の読み込みを実行できます。逆に *writev* システムコールは、複数の異なるバッファに対してアトミックな書き込みを実行できます。*read* や *write* によって行われるように、単一のバッファと長さをパラメータとして渡す代わりに、バッファと長さの配列へのポインタとそのサイズを渡します。

この機能により、
プロセスアドレス空間の異なる場所にあるバッファに対してアトミックな単一の書き込みを行え、
隣接するバッファにコピーする必要もありません。テープデバイスのように、それぞれの要求に対し、
テープブロックを出力をする必要があるようなレコードベースのデバイスを抽象化した場合、
アトミックな書き込みが必要になります。
また、単一の読み込みリクエストで複数のバッファに読み込めるのは非常に便利です
(たとえばレコードヘッダとデータをそれぞれ別のバッファに読み込む場合など)。
もちろん単一の大きなバッファにデータを読み込み、読み込んだデータを必要な場所に移動することで scatter 動作をシミュレートすることは可能です。ただし、このようなメモリ間のコピーのコストは、アプリケーションの動作に必要な時間を 2 倍以上にしてしまうことも良くあります。

send と *recv* がそれぞれ、*sendto* と *recvfrom* のライブラリインタフェースとして実装可能であったのと同じく、*read* と *write* をそれぞれ、*readv* と *writev* のライブラリインタフェースとして実装も可能であったでしょう。しかし、*read* と *write* はより頻繁に使われるため、シミュレートするための追加コストを考えるとライブラリインタフェースとしての実装は割に合わなかったでしょう。

2.1.6.6. 複数のファイルシステムのサポート

ネットワークコンピューティングの発達により、
ローカルおよびリモートファイルシステムへの対応が望まれるようになりました。
複数のファイルシステムのサポートを簡単にするために、開発者は *vnode* インタフェースをカーネルに追加しました。vnode インタフェースから提供される操作は、以前にローカルファイルシステムでサポートされていたファイルシステム操作とほぼ同じですが、
幅広いファイルシステムにより使用ができるようになっています。

- ローカルのディスクファイルシステム
- 各種リモートファイルシステムプロトコルによりインポートされたファイル
- 読み込み専用 CD-ROM ファイルシステム
- 特殊機能を提供するファイルシステム。たとえば /proc ファイルシステムなど

4.4BSD 由来の OS の中には FreeBSD のように、*mount*

でファイルシステムが初めて参照された時にファイルシステムを動的に読み込むことが
できるものもあります。 vnode インタフェースについては 6.5 節、 補助サポートルーチンについては 6.6 節、 特殊機能ファイルシステムについては 6.7 節に記載されています。

2.1.7. ファイルシステム

通常ファイルとは一次元のバイト列であり、 任意の場所から読み込み・書き込みが可能です。
カーネルは、ファイルのレコード境界を認識しませんが、 多くのプログラムは 改行 (LF) 文字を行の終りと認識します。
またこれとは異なるファイル構造を利用するアプリケーションもあります。
ファイル自身には、ファイルに関するシステム情報はまったく含まれません。
各ファイルのファイル所有者、許可属性、
使用状況などのいくつかの情報はファイルではなくファイルシステムが保持しています。

ファイル名は最大 255 文字までの文字列です。 ファイル名はディレクトリと呼ばれる型のファイルに保管されます。
ディレクトリに含まれるファイルの情報はディレクトリエントリと呼ばれ、
ファイル名以外にファイルそのものへのポインタも含みます。
ディレクトリエントリには通常のファイル以外に、 他のディレクトリを参照するエントリが含まれます。
このようにしてディレクトリとファイルによる階層が形作られ、 その階層構造をファイルシステムと呼びます。

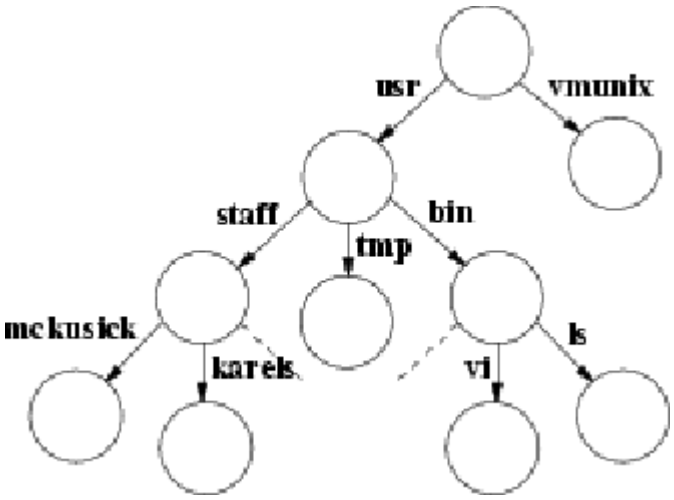

図 2. 小規模なファイルシステム

小規模なファイルシステムは小規模なファイルシステムの一例です。
ディレクトリはサブディレクトリを含むことができ、 入れ子の深さには特に制限はありません。
ファイルシステムの一貫性を保つため、
カーネルはプロセスが直接ディレクトリへ書き込むことを禁止しています。
ファイルシステムには、通常ファイル、ディレクトリ以外に、
デバイスファイルやソケットなどの他のオブジェクトへの参照も含まれます。

ファイルシステムは、 ルートディレクトリ を始点とする木構造を持っています。
ルートディレクトリは、 スラッシュ と呼ばれる場合もあり、斜線文字(/)で表されます。
ルートディレクトリにはファイルが含まれます。 図 2.2 の例では、 usr ディレクトリが含まれ、その usr ディレクトリには、 bin ディレクトリが含まれます。 bin ディレクトリには、通常 ls や vi をはじめとする、プログラムの実行可能コードが含まれます。

プロセスは、ファイルの指定をパス名によって行います。 パス名は、0 個以上のファイル名を斜線文字

(/)で区切った文字列です。 カーネルはパス名を解釈するため、それぞれのプロセスに 2 つのパス名を関連付けます。 プロセスのルートディレクトリは、プロセスがアクセスできるファイルシステム上で最も上位の点です。 通常、このルートディレクトリは、ファイルシステム全体のルートディレクトリに設定されます。 斜線文字 (/) ではじまるパス名は絶対パス名と呼ばれ、 カーネルは、そのパス名がプロセスのルートディレクトリから開始するものと解釈します。

斜線文字 (/) ではじまらないパス名は相対パス名と呼ばれ、 プロセスのカレント作業ディレクトリを基準とした相対的なパスとして解釈されます (このディレクトリは、短縮して カレントディレクトリ または、 作業ディレクトリ とも呼ばれます)。 カレントディレクトリそのものは、 ドット とも呼ばれ、1 つのピリオド (.) で表されます。 ファイル名 ドットドット (..) は、 ディレクトリの親ディレクトリを表します。 ルートディレクトリの親ディレクトリはルートディレクトリ自身です。

chroot システムコールにより、プロセスのルートディレクトリを、 *chdir* システムコールにより、カレントディレクトリを変更できます。 *chdir* はいつでも行えますが、 *chroot* の実行は、管理者特権を持つプロセスに限られます。 *chroot* は通常、システムに対するアクセス制限を課すために用いられます。

図 2.2 のファイルシステムにおいて、プロセスのルートディレクトリがファイルシステムのルートディレクトリで、カレントディレクトリが /usr であったとします。このとき、 vi を参照するには、絶対パスを用いて、 /usr/bin/vi とも書けますし、カレントディレクトリからの相対パスを用いて、 bin/vi とも書けます。

システムのユーティリティやデータベースは、 よく知られたある決まったディレクトリに保存されます。 ファイルシステムの階層構造としてよく知られたものに、 各々のユーザのホームディレクトリがあります。 たとえば、図 2.2 の /usr/staff/mckusick や /usr/staff/karels などです。 ユーザがログインすると、シェルのカレントディレクトリはホームディレクトリに設定されます。 ユーザはホームディレクトリ内で通常ファイルの作成と同様にディレクトリも作成できるため、 複雑な階層構造を構築することも可能です。

ユーザからはファイルシステムが 1 つに見えますが、 システムは 1 つの仮想ファイルシステムが、実際には異なるデバイス上の複数の物理ファイルシステムから構成されていることを認識しています。 物理ファイルシステムは、異なったデバイスにまたがることはできません。 ほとんどの場合、物理ディスクデバイスは複数の論理デバイスに分割されるため、 1 つの物理デバイス上に複数のファイルシステムを構成することもできます。 すべての絶対パス名を解決できるファイルシステムを ルートファイルシステムと呼び、常に利用可能な状態になっています。 他のファイルシステムは、マウントすることができます。 マウントとは、ルートファイルシステムのディレクトリ構造の一部として統合する操作です。 ファイルシステムにマウントされたディレクトリの参照は、 そのマウントされたファイルシステムのルートディレクトリの参照へと カーネルによって透過的に変換されます。

link システムコールは、既存のファイル名に、別名を与えます。 リンクが成功すると、ファイルはどちらのファイル名からでもアクセスできるようになります。 ファイル名は *unlink* システムコールにより削除できます。 ファイルを参照していた最後の名前が削除されると (さらにファイルを開いていた最後のプロセスがファイルを閉じると) ファイルそのものも削除されます。

ファイルはディレクトリ内で階層構造を持って保持されます。

ディレクトリそのものも一種のファイルですが、 ディレクトリは一般のファイルと異なり、
システムによって決められた構造を持っています。
ディレクトリは一般のファイルと同じくプロセスから読み込むことが可能ですが、
ディレクトリに変更を加えられるのはカーネルだけです。 ディレクトリは *mkdir*
システムコールで作成し、 *rmdir* システムコールで削除します。 4.2BSD 以前のシステムにおける *mkdir*
と *rmdir* システムコールは、一連の *link*、*unlink* システムコールの実行として実装されていました。
明示的にディレクトリの作成、 削除を行うシステムコールを新たに追加した理由は、3 つあります。

1. アトミックな動作を可能にするため。 link システムコールによる実装の場合と異なり、
   システムがクラッシュした場合に ディレクトリの構造が中途半端なままになることがありません。
2. ネットワークファイルシステムを使用している場合には シリアライズ (操作順序の保証)
   を行うため、ファイルおよびディレクトリの作成、 削除はアトミックに行われる必要があります。
3. UNIX 以外のファイルシステム (他のパーティション上の MS-DOS ファイルシステムなど)
   をサポートする場合、 そのファイルシステムが link
   システムコールをサポートしない可能性があります。
   たとえそれがディレクトリをサポートするファイルシステムであっても、 UNIX
   ファイルシステムとは異なり、ディレクトリをリンクとして作成、 削除しないものもあります。
   そのためそのようなファイルシステムでは、 ディレクトリの作成、削除は、
   明示的に要求されない限り行われません。

chown システムコールはファイルの所有者とグループを設定します。 *chmod*
システムコールは、ファイルの保護モードを変更します。 これらのファイルの属性は、 *stat*
システムコールをファイル名に対し実行することで読み出すことができます。 *fchown*、 *fchmod*、 *fstat*
システムコールは、同様な動作をファイル名ではなくファイル記述子に対して行います。 *rename*
システムコールは、ファイルに新しい名前をつけて古い名前を削除します。
ディレクトリ作成・削除操作と同じように、 *rename*
システムコールはローカルファイルシステムの名前変更動作をアトミックにするため 4.2BSD
で追加されました。 後に、この動作はネットワーク上の非 UNIX
ファイルシステムに対して名前変更操作を行う場合に有効であることがわかりました。

*truncate*は、 4.2BSD で追加されたファイルを任意の長さに切り詰めるシステムコールです。
このシステムコール追加の主な目的は
ランダムアクセスファイルの最後をプログラムが最後にアクセスした場所に設定する、 という動作を持つ
Fortran ランタイムライブラリのサポートでした。 *truncate* システムコールを使用しない場合、
ファイルの長さを縮める唯一の方法は必要な部分をコピーしたファイルを作成し、
元のファイルを削除した後にコピーしたファイルをリネームする方法です。
このアルゴリズムは遅いだけでなく、
空き容量の少ないファイルシステムでは失敗する可能性があります。

ファイルシステムにファイルを縮める機能が追加されると、
それはカーネルが大きな空のディレクトリを小さくする用途に使用するようになりました。
空のディレクトリを縮小すると、 ファイルの作成、
削除時にカーネルがファイルを検索する時間を短縮できるという利点があります。

新規に作成されたファイルには、
作成したプロセスのユーザ識別子と作成が行われたディレクトリのグループ識別子を与えられます。
ファイルの保護用に 3 レベルのアクセス制御機構が用意されています。 この 3
レベルのファイルアクセス許可は

1. ファイルを所有しているユーザ
2. ファイルを所有しているグループ
3. 他のすべて

に対して設定することができます。それぞれのアクセスレベルは、さらに読み取り許可、書き込み許可、実行許可に分けられています。

ファイルは作成時に長さが 0 であり、書き込みされるにつれて長くなっていきます。システムはファイルが開かれると、対応する記述子の現在位置を指定するポインタを保持します。このポインタはファイル内をランダムアクセスするように動かすことが可能です。*fork* や *dup* システムコールによりファイル記述子を共有するプロセス間では、この現在位置ポインタは共有されます。別々の *open* システムコールによって作成されたファイル記述子は、独立した現在位置ポインタを持ちます。ファイルは穴を持つことがあります。穴とはファイルの一次元構造の中で、データが一度も書き込まれたことのない空の部分です。ファイルの最後尾より後にポインタを動かし書き込みを行うことで、ファイルに穴をつくることができます。読み込まれた場合、穴は 0 の値をもつバイトとして扱われます。

初期の UNIX システムではファイル名に 14 文字以内という制限があり、よく問題となっていました。たとえば、ユーザは当然ながら長く説明的なファイル名を付けたいと望みますし、basename.extension という慣用的なファイル命名規則を考えると、extension (C のソースファイルの `.c`、中間バイナリオブジェクトファイルの `.o` というように、ファイルの種類を示す部分) に 1 から 3 文字必要ですから、basename に付けられる文字数は 10 から 12 しか残っていません。ソースコード制御システムやエディタは通常、独自の目的のためにさらに 2 文字をファイル名の前後に付加しますので、実際に使えるのは、8 から 10 文字になります。basename として英語を一単語 (たとえば multiplexer) 使うだけで、簡単に 10 から 12 文字になってしまうでしょう。

このような制限を守るのは不可能ではありませんが、危険な場合もあります。他の UNIX システムでは、より長いファイル名を受け付けるものの実際にファイルを作成する時点でファイル名を切り詰めるものがあるからです。C のソースコードファイル multiplexer.c (すでに 13 文字です) のソースコード制御ファイルは、頭に `s.` が付加されて s.multiplexer となります。このファイルは、C ソースの文書の `troff` ソースファイル multiplexer.ms のソースコード制御ファイルと区別がつきません。ソースコード制御システムはこの問題に対して警告を出さないため、これらの 2 つのファイル内容の取り違えは容易に発生します。注意深くコーディングすればこのような問題は避けられますが、4.2BSD でロングファイルネームが導入されたことで この問題は実質的になくなりました。

2.1.8. ファイル記録機構

ローカルファイルシステムに対する操作には二種類あります。まず、ローカルファイルシステムすべてに共通して階層化されたファイルのネーミング、ロック、割り当て、属性管理、保護といった、データの記録方法とは独立した機能です。4.4BSD はこれらの機能を提供する単一の実装を備えています。

もう一つは記録媒体上におけるデータ構成と管理です。ファイル内容を記録媒体上に配置するのはファイル記録機構の役割であり、4.4BSD は 3

種類の異なるファイル配置法に対応しています。

- 伝統的な Berkeley Fast Filesystem
- Sprite という OS の設計に由来する ログ構造化ファイルシステム Rosenblum & Ousterhout, 1992
- メモリベースのファイルシステム

これらのファイル記録機構の構成はまったく異なるものですが、
それを使用するプロセスからは違いを意識することはありません。

Fast Filesystem は、データをシリンダグループという単位で構成します。
ファイルシステム階層の配置から考えて同時にアクセスされやすいと考えられるファイルは、
同じシリンダグループに記録され、同時にアクセスされる可能性の低いファイルは
異なるシリンダグループに記録されます。
この記録機構では以上のように、複数のファイルが同時に書き込まれたとしても、
記録される場所はディスクのまったく違う場所になる可能性があるのです。

ログ構造化ファイルシステムは、データをログという形で構成します。
ある時点で記録されたデータはすべて一つに集められ、 同じディスクの場所に書き込まれます。
データが上書きされることは絶対にありません。 ファイルの更新は、ファイルを上書きする代わりに
新しいファイルを書き込んでそのファイルを置き換えることによって行なわれます。
ファイルシステムに空き容量がなくなり新たに空き容量が必要になった場合は ゴミ集め (garbage-collection) プロセスが実行され、 古いファイルが再利用されます。

メモリベースのファイルシステムは、 データを仮想メモリに記録するように設計されたものです。
これは /tmp のように高速アクセスが必要で、 永続的でないファイルシステムに使われます。
メモリベースファイルシステムの目標は、
仮想メモリ資源の利用量を可能な限り最小限に保つことにあります。

2.1.9. ネットワークファイルシステム

当初、ネットワーク通信はデータをあるマシンから 他のマシンへ転送するために用いられていましたが、
のちにそれは、ユーザが離れたマシンへログイン可能な形に発展しました。
次に期待されたのはユーザがデータを取り行くのではなく、
ユーザの元にデータがやってくるようにすることでした。
そのために生まれたのがネットワークファイルシステムです。
ローカルで作業しているユーザはキー入力時にネットワークの遅れを感じず、
より応答性の良い環境を手に入れたのです。

ファイルシステムをローカルマシンに持ってくることは 初期のサーバ-クライアント型アプリケーションの中で主要なもののひとつでした。 サーバ は 1 つもしくはそれ以上のファイルシステムを エクスポートするリモートのマシンです。 クライアントはそのファイルシステムをインポートするローカルのマシンです。 ローカルのクライアントから見ると、リモートでマウントされたファイルシステムは ローカルにマウントされた他のファイルシステムのようにファイルツリーの名前空間に現れます。 ローカルのクライアントは
リモートのファイルシステム上にディレクトリを変えたり、
ローカルのファイルシステム上でするのとまったく同じように
リモートのファイルシステムで読み書きをしたり、 バイナリを実行したりできます。

ローカルのクライアントが リモートのファイルシステム上で操作すると、
その操作要求がひとまとめにされてサーバに送られます。 サーバは要求された操作を行い、

クライアントから要求された情報、もしくは、 なぜその要求が拒絶されたかを示すエラーを返します。
適切な性能を得るには、
クライアントは頻繁にアクセスされたデータをキャッシュしなければなりません。
リモートファイルシステムの複雑さは、
サーバと多くのクライアントの間のキャッシュの一貫性を維持することにあります。

長期にわたって数多くのリモートファイルシステムプロトコルが開発されてきましたが、 UNIX システムにおいて最も普及しているものは、 そのプロトコルと実装の大部分が Sun Microsystems によって行なわれた ネットワークファイルシステム (NFS) です。
実装はプロトコル規格から独立して行われましたが、 4.4BSD カーネルは NFS プロトコルをサポートしています Macklem, 1994。 NFS プロトコルについては 9 章で説明しています。

2.1.10. 端末

端末は、標準的なシステム I/O 操作はもちろんのこと、
入力文字の編集や出力のディレイの制御をするための端末固有の操作をひととおり サポートしています。
一番低いレベルにあるのは、ハードウェア端末ポートを制御する端末デバイスドライバです。
端末入力は、たとえばボーレートのような基礎的な通信特性や、
パリティ検査のようなソフトウェアで制御可能なパラメータ類に 従って扱われます。

端末デバイスドライバの上の層には、 文字処理をどの程度行なうかを定義している ラインディシプリン (line discipline; 回線端末制御) と呼ばれるものがあります。 対話的なログインをするためにポートが用いられるときにはデフォルトのラインディシプリンが選択され、
そのラインディシプリンはカノニカルモードで動作します。
これは、入力が標準的な行指向編集機能を提供するように処理され、
入力自体を行単位の処理で表現するモードです。

スクリーンエディタや、他のコンピュータと通信をするプログラム (訳注: telnet など) は、普通非カノニカルモード (*raw* モードやキャラクタごとのモード *(character-at-a-time mode)* などとも呼ばれます) で動作します。 これらのモードでは、入力はそのまますぐに読み込み側のプロセスへと渡されます。 すべての特殊文字入力の処理は無効化されていて、
削除やその他の行編集処理は行なわれず、
すべての文字はその端末から読み込もうとしているプロセスへと渡されます。

端末は、この二つの両極端のモードの中間の多くの組み合わせで 設定することが可能です。
たとえば、あるスクリーンエディタがユーザからの割り込みを非同期的に
受け入れたい場合に、シグナルを生成する文字や出力の流量制御を許可したまま、
それ以外を非カノニカルモードで動かして、これらの文字以外の文字を
まったく解釈しないまま渡す、ということが可能です。

出力では、端末処理は次のような単純な整形サービスを提供しています。

- ラインフィードをキャリッジリターンとラインフィードの並びへと変換
- 特定の標準的な制御文字の後にディレイを挿入
- タブ文字の展開
- エコーされた非表示アスキー文字を `^C` (すなわち、アスキーのキャレット文字の後に、そのキャラクタの値をアスキーの `@` 文字からのオフセットとした アスキー文字) という二文字の並びとして表示

これらの整形機能は、コントロールリクエストを使ってそれぞれ独立に 無効化することが可能です。

2.1.11. プロセス間通信 (IPC)

4.4BSD のプロセス間通信 (IPC) は、コミュニケーションドメイン内で働くようになっています。現在サポートされているドメインには、同じマシン上で実行している複数のプロセス間 での通信用のローカルドメイン、TCP/IP プロトコルスイート用の (おそらく the Internet 内) インターネットドメイン、 ISO/OSI プロトコルファミリでの通信を行なうことが必要なサイト間通信用の ISO/OSI プロトコルファミリ、XEROX Network Systems (XNS) を使用したプロセス間通信用の *XNS* ドメインが含まれています。

ドメイン内では、ソケットとして知られ ている通信終端間で通信が行なわれます。 2.6 節で説明しているように、 *socket* システムコールはソケットを生成し、その記述子を返します。 他の IPC システムコールについては 11 章で解説します。
各ソケットは、通信セマンティクスを定義した型を持ちます。
このセマンティクスには信頼性、順序、メッセージの重複防止が 含まれています。

各ソケットは、通信プロトコル と関連しています。
ここでのプロトコルは、通信相手のソケットの型に従って
そのソケットで要求されているセマンティクスを提供します。
アプリケーションは、ソケットを生成する際に特定のプロトコルを
要求することができますし、また、そのシステムは、将来生成される
ソケットの型にふさわしいプロトコルを選択するようにすることも 可能です。

ソケットは、そのソケットと関連づけされた (バインドされた) アドレスを持つことができます。
ソケットアドレスの形式と意味は、そのソケットが生成された
コミュニケーションドメインに依存します。
ローカルドメインにおいてソケットに名前をバインドすると、
そのファイルシステムにおいてファイルが生成されます。

ソケットを通じて送受信される通常のデータは型づけされていません。
データ表現については、プロセス間通信機能の最上位に位置するライブラリに責任があります。
通常データの配送に加えて、コミュニケーションドメインは *access rights* という特別な型のデータの送受信をサポートすることができます。 たとえばローカルドメインはプロセス間で記述子を渡すために、この機能を使用します。

4.2BSD より前の UNIX におけるネットワーク機能の実装は、大抵キャラクタデバイスインタフェースをオーバロードさせることで 動作していました。
ソケットインタフェースの目的の一つは、単純なプログラムが
ストリーム型の通信を変更せずに動作するようにすることです。 そのようなプログラムは、*read* と *write* のシステムコールが変更されなければ 動作します。 当然、元のインタフェースがそのまま残されれば、ストリーム型のソケット上で動作し続けるようになります。 *send* の各呼び出しで指定しなければならない 送信先アドレスを持つデータグラムを送信するような、より複雑なソケット用に新しいインタフェースが追加されました。

もう一つの利点は、この新しいインタフェースは移植性が 非常に良いということです。バークレーから入手できたテストリリースのすぐ後で、 ソケットインタフェースは UNIX ベンダによって System III に移植されました (しかし、AT&T は System V Release 4 のリリースまでソケットインタフェースをサポートせず、 その代わりに Eighth Edition

のストリーム機構を使用する ことを決めました)。 ソケットインタフェースはまた、Excelan 社や Interlan 社のような ベンダによって多くのイーサネットカードで動作するように移植され、マシンが小さすぎてメインプロセッサ中でネットワーク通信を動作 させることができない PC 市場に売り出されました。 ごく最近では、Microsoft 社の Windows 用の Winsock ネットワークインタフェースの基盤として ソケットインタフェースが使われています。

2.1.12. ネットワーク通信

ソケット IPC 機構が対応している ネットワークドメインのいくつかは、ネットワークプロトコルへのアクセスを提供しています。 これらのプロトコルは理論上、カーネルのソケットソフトウェアよりも下の層にある 別のソフトウェアとして実装されています。カーネルは、バッファ管理、メッセージ配送、 各プロトコルへの汎用インタフェースを提供し、また、さまざまなネットワークプロトコルを使用するための
ネットワークインタフェースドライバへのインタフェースなど、 多くの付随サービスを提供します。

4.2BSD が実装された時点ではさまざまなネットワークプロトコルが使用され、また開発中の段階にありました。 それらはそれぞれ固有の強みと弱みを持っており、明らかに優れたプロトコルやプロトコルスイートというものは存在しませんでした。 4.2BSD は複数のプロトコルに対応することで、 バークレー校の環境で利用可能だったさまざまなマシン間での資源の共有や、相互運用の提供を可能にしていました。 またこの複数プロトコルへの対応は、将来的な変更に備えて設計されていました。 今日利用されている 10-100Mbps のイーサネット用のプロトコルは、 将来の 1-10Gbps 光ファイバネットワークに対して、おそらく十分なものではないでしょう。
そのため、ネットワーク通信レイヤは複数のプロトコルに対応できるように 設計されています。
新しいプロトコルがカーネルに追加されても、
既存のプロトコルがその影響を受けることはまったくありません。
新しいアプリケーションは新しいネットワークプロトコルで動作し、
一方で既存のアプリケーションもまた、 それと同じ物理ネットワーク上で今までどおりのプロトコルを利用し続けることが可能です。

2.1.13. ネットワーク実装

4.2BSD で実装された最初のプロトコルスイートは DARPA の Transmission Control Protocol/Internet Protocol (TCP/IP) でした。 CSRG は、ソケット IPC フレームワークに組み込む最初のネットワークとして TCP/IP を選択しました。その理由は、4.1BSD ベースの実装が DARPA がスポンサーとなっていた Bolt、Beranek、Newman (BBN) におけるプロジェクトからパブリックに入手可能だったからです。それは大きな選択でした。 このプロトコルスイートが非常に広く利用されたのは、 主に 4.2BSD でのこの実装が理由となっています。 TCP/IP の実装に対するその後の性能と能力の改善も広く採用されました。 TCP/IP の実装については、13 章で詳細に解説しています。

4.3BSD のリリースでは、メリーランド大学とコーネル大学で部分的に 開発された Xerox Network Systems (XNS) プロトコルスイートが追加されました。 このプロトコルスイートは、TCP/IP を使用して通信できない孤立したマシンと 通信するのに必要でした。

4.4BSD のリリースでは、近頃米国内外で増加している ISO プロトコルスイートが追加されました。 ISO プロトコル群のために多少異なるセマンティクスを定義したので、
これらのセマンティクスに適合させるため

ソケットインタフェースにいくつかの小さな変更が必要となりました。 その変更は、他の既存プロトコルのクライアントには分からないようになされています。 ISO プロトコル群用に 4.3BSD カーネルで提供された 2 レベルルーティングテーブルに対する大規模な追加も必要でした。 4.4BSD で大きく拡張されたルーティング機能は、 可変長アドレスとネットワークマスクを持つ任意のレベルのルーティングが含まれています。

2.1.14. システム運用

ブートストラップ機構はシステムを起動するために利用されます。 まず最初に、4.4BSD カーネルは CPU のメインメモリに読み込まれます。 カーネルが読み込まれると、特定の状態へハードウェアを設定する初期化フェーズに移行します。 次に、カーネルは自動設定 (autoconfiguration) を行ないます。 これは CPU に接続された周辺機器の検出と設定を行なう過程です。 システムは最初、ディスクチェック、アカウント処理、 quota チェックを行なうスタートアップスクリプトを シングルユーザモードで実行します。 スタートアップスクリプトは最後に一般的に利用されるシステムサービス群を起動し、システムを完全なマルチユーザモードに移行させます。

マルチユーザモードでは、プロセスが ユーザがアクセスできるように設定された端末回線やネットワークポート上でのログイン要求を待ちます。 ログイン要求が検出されるとログインプロセスが生成され、 ユーザの確認処理が行われます。 そしてユーザの確認処理が成功すると、 そのユーザに対して、他のプロセスを実行できるようにするためのログインシェルが生成されます。

参考文献

Accetta et al, 1986 Mach: A New Kernel Foundation for UNIX Development" M.Accetta R.Baron W.Bolosky D.Golub R.Rashid A.Tevanian M.Young 93-113 USENIX Association Conference Proceedings USENIX Association June 1986

Cheriton, 1988 The V Distributed System D. R.Cheriton 314-333 Comm ACM, 31, 3 March 1988

Ewens et al, 1985 Tunis: A Distributed Multiprocessor Operating System P.Ewens D. R.Blythe M.Funkenhauser R. C.Holt 247-254 USENIX Assocation Conference Proceedings USENIX Association June 1985

Gingell et al, 1987 Virtual Memory Architecture in SunOS R.Gingell J.Moran W.Shannon 81-94 USENIX Association Conference Proceedings USENIX Association June 1987

Kernighan & Pike, 1984 The UNIX Programming Environment B. W.Kernighan R.Pike Prentice-Hall Englewood Cliffs NJ 1984

Macklem, 1994 The 4.4BSD NFS Implementation R.Macklem 6:1-14 4.4BSD System Manager’s Manual O’Reilly & Associates, Inc. Sebastopol CA 1994

McKusick & Karels, 1988 Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel M. K.McKusick M. J.Karels 295-304 USENIX Assocation Conference Proceedings USENIX Assocation June 1998

McKusick et al, 1994 Berkeley Software Architecture Manual, 4.4BSD Edition M. K.McKusick M. J.Karels S. J.Leffler W. N.Joy R. S.Faber 5:1-42 4.4BSD Programmer’s Supplementary Documents

O'Reilly & Associates, Inc. Sebastopol CA 1994

Ritchie, 1988 Early Kernel Design private communication D. M.Ritchie March 1988

Rosenblum & Ousterhout, 1992 The Design and Implementation of a Log-Structured File System M.Rosenblum K.Ousterhout 26-52 ACM Transactions on Computer Systems, 10, 1 Association for Computing Machinery February 1992

Rozier et al, 1988 Chorus Distributed Operating Systems M.Rozier V.Abrossimov F.Armand I.Boule M.Gien M.Guillemont F.Herrmann C.Kaiser S.Langlois P.Leonard W.Neuhauser 305-370 USENIX Computing Systems, 1, 4 Fall 1988

Tevanian, 1987 Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach Technical Report CMU-CS-88-106, A.Tevanian Department of Computer Science, Carnegie-Mellon University Pittsburgh PA December 1987

付録 A: 日本語化について

The Design and Implementation of 4.4BSD Operating System Chapter 2 の日本語化は、原著の出版元である Addison-Weslay、翻訳出版権を保有する Pearson Education Japan の協力を得て、FreeBSD 日本語ドキュメンテーションプロジェクト (FreeBSD doc-jp) によって行なわれました。日本語版について何かお気付きの点がありましたら日本語ドキュメンテーションプロジェクト doc-jp@jp.FreeBSD.org までご連絡ください。

2001 年 5 月 5 日にスタートした日本語化作業には、さまざまな方々が翻訳に参加されました。FreeBSD doc-jp では、FreeBSD 関連文書の日本語版を作成する作業を精力的に続けています。この作業に協力したいと思われる方は、ぜひFreeBSD 日本語ドキュメンテーションプロジェクトのページをご覧の上 doc-jp へご参加ください。

翻訳者

- 杉村 貴士 sugimura@jp.FreeBSD.org (2.1, 2.2 節)
- IKENO Naoki nao@mc.kcom.ne.jp (2.3 節)
- 田畑 喜晃 ytabata@tkf.att.ne.jp (2.4 節)
- Atsuto atsuto@guitar.interq.or.jp (2.5 節)
- はらだきろうkiroh@jp.FreeBSD.org (2.6, 2.7 節)
- 高田 知樹 tomoki@leergirls.org (2.8 節)
- 倉品 英行 rushani@bl.mmtr.or.jp (2.9 節)
- 塩崎 拓也 tshiozak@FreeBSD.org (2.10 節)
- こが よういちろうy-koga@jp.FreeBSD.org (2.11, 2.13 節)
- 森 直之mori@jp.FreeBSD.org (2.12 節)

- 坂井 順行 sakai@lac.co.jp (2.14 節)
- 内川 喜章 yoshiaki@kt.rim.or.jp (査読)
- 日野 浩志 hino@ccm.cl.nec.co.jp (査読)
- 山口 雅信 yamagu-m@titan.ocn.ne.jp (翻訳提供)